Word-length Optimization Beyond Straight Line Code

David Boland and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
{david.boland03, g.constantinides}@imperial.ac.uk

ABSTRACT

The silicon area benefits that result from word-length optimization have been widely reported by the FPGA community. However, to date, most approaches are restricted to straight line code, or code that can be converted into straight line code using techniques such as loop-unrolling. In this paper, we take the first steps towards creating analytical techniques to optimize the precision used throughout custom FPGA accelerators for algorithms that contain loops with data dependent exit conditions. To achieve this, we build on ideas emanating from the software verification community to prove program termination. Our idea is to apply word-length optimization techniques to find the minimum precision required to guarantee that a loop with data dependent exit conditions will terminate.

Without techniques to analyze algorithms containing these types of loops, a hardware designer may elect to implement every arithmetic operator throughout a custom FPGA-based accelerator using IEEE-754 standard single or double precision arithmetic. With this approach, the FPGA accelerator would have comparable accuracy to a software implementation. However, we show that using our new technique to create custom fixed and floating point designs, we can obtain silicon area savings of up to 50% over IEEE standard single precision arithmetic, or 80% over IEEE standard double precision arithmetic, at the same time as providing guarantees that the created hardware designs will work in practice.

Categories and Subject Descriptors

G.1.0 [**NUMERICAL ANALYSIS**]: Miscellaneous

General Terms

Algorithms, Design, Verification.

Keywords

Precision Analysis, Loop Termination, Word-length Optimization.

1. INTRODUCTION

In recent years we have seen an explosion in the use of embedded devices throughout everyday life. These devices demand high performance with minimal power use. FPGAs are often proposed as an alternative to create designs that exhibit much greater energy efficiency than a software counterpart. To obtain this efficiency, it is imperative that the available silicon area is used effectively; this means not performing any unnecessary computation. This ideology should not only apply at an algorithmic level, but also at a much finer arithmetic level. Word-length optimization is a key technique to ensure that every individual bit within an arithmetic operation is computing useful information.

The underlying concept behind word-length optimization is that error is inherent in most numerical algorithms because it is either impossible or infeasible to represent numbers exactly. This means that almost every algorithm operating on real numbers must be tolerant of errors to some extent. Ideally one should choose the minimum precision throughout a design according to this tolerance because this will use the minimum silicon area. This in turn will reduce the power requirements, or allow room for additional parallelism.

A large variety of techniques have been proposed which attempt to realize these performance benefits, as we will discuss in Section 2. To date, these are based upon analyzing the finite precision errors that arise from every individual arithmetic operation and computing bounds on the range or relative error of the output. This information is used to determine the minimum precision required to meet an error specification. Unfortunately this process breaks down in the case of loops with data dependent exit conditions. This is because it is impossible to analyze every arithmetic operation *a priori* since it is unknown how many operations will actually occur.

In this paper, we assume that the loop termination condition embodies the algorithm designer's objective, such as convergence to a desired result. Consequently, we propose that the precision used throughout a custom FPGA design should be the minimum required to ensure that the loop will still terminate. This would guarantee that the hardware would work as desired.

In order to provide such guarantees, we build on concepts to prove program termination described by the software verification community. Firstly, we introduce techniques to model finite precision errors (both fixed and floating point) within a termination argument. We then demonstrate how it is possible to use analysis techniques from the field of word-length optimization to check the validity of this argument for various word-length specifications. We can then select the minimum precision that ensures the loop will terminate. A summary of the main contributions of this work is as follows:

- A method to incorporate finite precision models typically used in numerical analysis into a termination argument.
- A discussion of how to use precision analysis techniques to validate such a termination argument. These techniques are

FPGA'13, February 11–13, 2013, Monterey, California, USA.

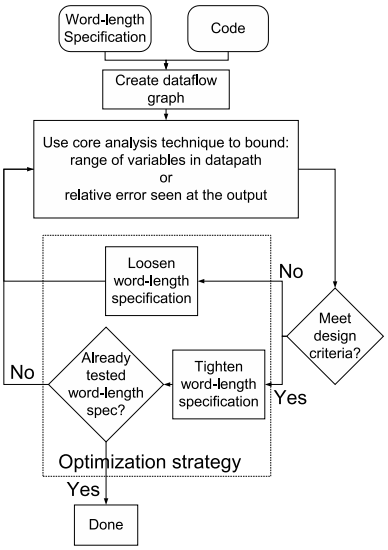


Figure 1: Overview of word-length optimization.

applicable where the loop body consists of the basic algebraic operations or smooth elementary functions. This takes it beyond the scope of existing termination proving tools.

- The integration of our analysis technique within a basic word-length optimization framework to illustrate the power of our tool in saving silicon area when creating custom FPGA designs.

We first introduce the reader to the field of word-length optimization in order to highlight how the analysis technique we describe could have a major impact on this body of work. We then walk through the basics of program termination, discuss the state of the art tools to prove termination and their limitations when seen from an FPGA-based computing point of view. After this, through the use of a few simple examples, we introduce our solution to utilize analysis techniques from the field of word-length optimization to prove program termination in the presence of finite precision errors. Finally, we demonstrate how our method can be used to tune the precision throughout an algorithm using four case studies.

2. WORD-LENGTH OPTIMIZATION

Figure 1 presents a high-level view of word-length optimization. The typical process first involves constructing a dataflow graph for the desired code and analyzing the error seen at the output of this graph for a given word-length specification. It then iteratively refines this word-length specification to create the smallest, lowest power, or fastest datapath that satisfies the desired design criterion. To date, the focus of word-length optimization frameworks has either been on the high-level optimization strategy to perform the word-length refinement, or the core analysis technique to compute bounds on the range or relative error of the output variables of a datapath over a user-provided input range. In this section, we discuss the main contributions in each of these areas.

Computing bounds on the range of variables in a datapath enables a user to choose the position of the MSB in a fixed point representation, or the number of bits required for the exponent in a floating point representation. Bounds on the relative error seen at the output of a datapath are important when computing metrics such as the signal to quantization noise ratio, or more generally to ensure the hardware datapath achieves the desired quality of output. Consequently, several core analysis techniques have been described to compute these bounds. As well as Monte-Carlo simulation, these include analytical techniques such as interval arithmetic, linear-time invariant system analysis, affine arithmetic, algebraic analysis and satisfiability modulo theory [10]. While simulation can potentially detect counter examples demonstrating that a word-length specification violates the design criteria, it runs the risk of not allocating sufficient bits because it is impractical to exhaustively simulate over a range of input data. In contrast, the analytical methods can provide guarantees that any word-length specification is appropriate. Unfortunately they will typically over-allocate bits because they are unable to calculate the optimal bounds. The existing tools trade quality of bounds for execution time and scalability of the procedure, as discussed in detail in [2]. Furthermore, various contributions have proposed slightly higher level strategies to help improve the quality of bounds and execution time of the core analysis techniques. For example, allowing expert hints to help improve the analysis [15] or applying some form of combination of these techniques to make the most of their contrasting strengths [17, 21, 22, 24].

The high-level optimization strategies which utilize these core analysis techniques seek to quickly find the best word-length specification that meets the design criterion. This is a complex problem because it has been shown that a variable word-length specification can obtain superior performance-error trade-offs than a uniform word-length specification [8]. Since an exhaustive search is generally impractical, various approaches have been proposed to quickly search for an optimal design. These include integer linear programming [9], simulated annealing [21] and various custom heuristics [8, 20, 29]. Alternatively, techniques have been suggested to reduce the search space by grouping signals to take into account resource sharing [4, 5], or guide the search by applying area and error models for adders and multipliers or other operators [6, 21]. Finally, research has also investigated whether to implement operators using fixed point, floating point or a some combination of the two approaches [19].

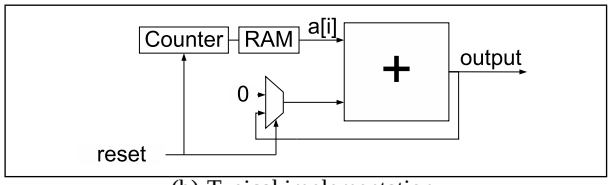
However, the fundamental limitation of these analytical word-length optimization techniques is that they need to know exactly how the data flows through the hardware. This means this body of work can only be applied to algorithms where the input code can be converted into static single assignment (SSA) form. Algorithms with conditional statements can be expanded into SSA by exploring each path separately. For example, for the code of Figure 2, the result of a/b lies in the interval $[0.1; 10]$, and the result of $a \times b$ lies in the interval $[1000; 10000]$; consequently, the variable $output$ lies somewhere in their union *i.e.* in the interval $[0.1; 10000]$. Similarly, any graph with feedback can be unrolled into SSA form provided the number of iterations is known *a priori*. From the perspective of computing the range or worst-case error of the variable $output$ from the code shown in Figure 3, the hardware architectures of Figures 3(b) and 3(c) are equivalent. Using the latter architecture, the framework of Figure 1 is still appropriate.

```
Function absolute_dif(a, b)
//a, b are known to lie in the interval [10; 100];
if a > b then
   output = a/b;
else
   output = a × b;
end if
```

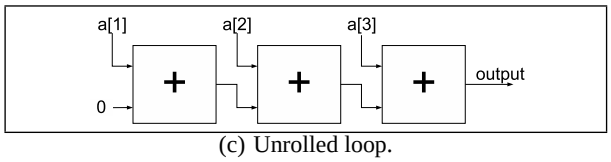
Figure 2: Expanding a conditional statement.

```
Vector_sum(a);
output = 0;
for i = 1; i <= 3; i++ do
   output = output+a[i];
end for
```

(a) Basic loop.



(b) Typical implementation.



(c) Unrolled loop.

Figure 3: Unrolling a for loop.

However, it is not possible to construct a finite cycle free graph for a loop with data-dependent exit conditions. While it is possible to analyze a single loop iteration, one cannot compute the bound on relative error seen at the final output because any errors will be amplified by feedback [3]. This means that none of the existing analytical word-length optimization tools can be used on these loops, while any simulation based word-length optimization tool will still potentially suffer from under-allocating bits and potentially create unsafe hardware.

To illustrate the difficulty of allocating bits within such a loop, consider the simple example of designing a dedicated hardware accelerator to compute the time for a radioactive substance to decay to a tolerable level. It is specified that the inputs for the rate of decay and tolerable limit are 4-bit values lying between $\frac{1}{16}$ and $\frac{15}{16}$, and the output number of years should be an 8-bit integer. Figure 4 shows the algorithm and a potential hardware implementation for this specification is given in Figure 5. The only unknown is the width of the internal variable $currAmount$. None of the existing word-length optimization tools are capable of choosing the precision of this variable. Suppose we were to arbitrarily choose this precision to be 5 bits lying between $\frac{1}{32}$ and 1. In this case, if the decayRate was $\frac{15}{16}$ and the limit was $\frac{1}{16}$, the algorithm would execute as in Table 1 and would never terminate!

```
Function Halflife(fixed4bit(decayRate), fixed4bit(limit))
int8 years = 0;
fixed?bit currAmount = 1;
while currAmount > limit do
   years = years+1;
   currAmount = currAmount × decayRate;
end while
return(years);
```

Figure 4: Basic function to compute time for radioactive substance to decay to a tolerable level.

Since loop termination is a necessary condition for correctness, this paper proposes to choose the precision such that the loop is guaranteed to terminate. For this case study, because the input has a very limited range, it is possible to exhaustively simulate until the minimum precision that will guarantee termination is computed. However, for wide input ranges, the execution time prevents such an approach. Instead, we discuss how ideas to prove loop termination can be utilized to allocate bits within such loops.

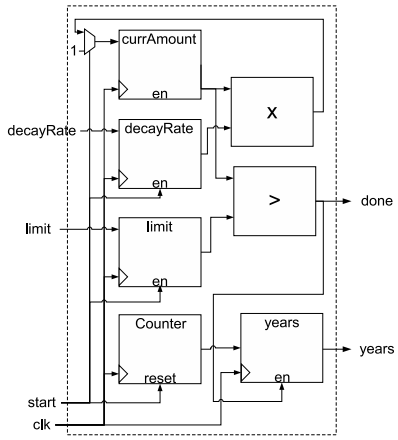


Figure 5: Hardware accelerator for halflife example.

Table 1: Execution of half-life algorithm with currAmount using 5 bits lying between $\frac{1}{32}$ and 1.

Loop Iteration	Years	currAmount (before rounding)	currAmount (after rounding)
0	0	1	1
1	1	0.9375	$^{30}/_{32}$
2	2	0.87890625	$^{28}/_{32}$
3	3	0.87890625	$^{26}/_{32}$
4	4	0.76171875	$^{24}/_{32}$
5	5	0.703125	$^{23}/_{32}$
⋮	⋮	⋮	⋮
19	19	0.263671875	$^{9}/_{32}$
20	20	0.234375	$^{8}/_{32}$
21	21	0.234375	$^{8}/_{32}$

Tools to automatically generate termination proofs have drawn interest from several communities ever since their use was first demonstrated to find software bugs in real systems code [12]. Since then, many potential further uses of these tools have been identified [13]. In this paper, we seek to extend the use of these tools by making them applicable for optimizing the precision throughout a custom hardware accelerator for numerical algorithms. The next section provides an introduction to the relevant concepts that we have adopted from this community for the unfamiliar reader before discussing the state-of-the-art in termination tools. In Section 4, we then discuss our suggested method that builds on these ideas and enables us to choose the number of bits required in an algorithm with data dependent loop exit conditions.

3. PROVING TERMINATION

The main concept we have adopted from termination provers is that of ranking functions. After introducing some notation, we will discuss this technique through the use of some simple examples.

3.1 Notation

For a loop containing n variables, we label the loop variables x_1 to x_n before the loop update, and x'_1 to x'_n after the loop update. We focus on analyzing loops that follow the structure described by Figure 6. These are singly nested loops which contain bounded input variables, linear and conjunctive loop exit conditions and transition statements that consist of the basic algebraic operations $(+, -, \times, /)$ or smooth elementary functions.

```
// Loop input variables with defined input ranges
x_1 = 100 × rand();
x_2 = 5 − 10 × rand();
while ((x_1 > 0)&&(x_2 < 1000))do //Loop exit conditions
    //Loop transition statements
    x_1 = x_1 + 1
    x_2 = x_2 × x_1
end while
```

Figure 6: Overview of loop structure.

3.2 Ranking functions

Proving program termination using a ranking function is based on the following steps:

1. Construct a ranking function $f(x_1, ..., x_n)$ that maps every potential state within the loop to a positive real number.
2. Prove that for all potential values of the variables $x_1, ..., x_n$ within the loop body, when the ranking function is applied to the loop variables before and after the loop transition statements, it always decreases by more than some fixed amount $\epsilon > 0$, *i.e.* $f(x'_1, ..., x'_n) < f(x_1, ..., x_n) - \epsilon$.

The reason this proves program termination, is because if the ranking function always decreases by at least the fixed amount ϵ, eventually there will be a loop transition such that $f(x'_1, ..., x'_n) \leq 0$. This corresponds to the loop terminating.

To illustrate this process, we discuss the termination argument for the three examples in Figure 7:

1. (a) Choose a ranking function that maps every state in the loop to a positive real number: $f(i) = i$ is one such function.
   (b) Apply ranking function to loop variables before and after the loop transition statements: $f(i) = i$, $f(i') = i - 1$.
   (c) Prove that $f(i') < f(i)$: $0 < 1 \rightarrow i - 1 < i$.
2. (a) Choose a ranking function that maps every state in the loop to a positive real number: $f(i) = 100 - i$ maps every potential state to the set of integers between 0 and maxInt.
   (b) Apply ranking function to loop variables before and after the loop transition statements: $f(i) = 100 - i$, $f(i') = 100 - (i + 1)$.
   (c) Prove that $f(i') < f(i)$: $0 < 1 \rightarrow 100 - (i + 1) < 100 - i$.
3. (a) Choose a ranking function that maps every state in the loop to a positive real number: $f(i, j, k) = maxInt + 100 - i - j$ maps every potential state to the set of integers between 0 and $2\times$ maxInt+100.

```
int i;
while i > 0 do
    i = i-1;
end while
```
(a) Case study 1.

```
int i;
while i < 100 do
    i = i+1;
end while
```
(b) Case study 2.

```
int i,j,k;
while (i < 100)&&(j < k) do
    temp = i;
    i = j;
    j = temp+1;
    k = k-1;
end while
```
(c) Case study 3.

Figure 7: Termination case studies.

   (b) Apply ranking function to loop variables before and after the loop transition statements: $f(i, j, k) = maxInt + 100 - i - j$, $f(i', j', k') = maxInt + 100 - j - (i+1)$.
   (c) Prove that $f(i', j', k') < f(i, j, k)$: $0 < 1 \rightarrow maxInt + 100 - i - j < maxInt + 100 - j - (i + 1)$.

While it is trivial to demonstrate termination in these benchmarks, for more complex problems, constructing a suitable ranking function and proving that the ranking function argument holds are both difficult tasks. As a result it is desirable to create automated analysis techniques. Any such process must take care when dealing with finite precision number systems.

A trivial example of why it is important to consider finite precision error can be seen in Figure 8, which once again examines the case study in Figure 7(a), but assumes a floating point representation is chosen for i. We have shown that this program will terminate under infinite precision. Similarly, the program will terminate for any fixed point number system containing a unit digit. However, the code will not necessarily terminate for a floating point number system, because the decrement can be counteracted by round-off error. To demonstrate this, suppose we had a custom number system of a 4 bit exponent with no bias and a 5 bit mantissa, and our initial value of i was 256. In this loop, we would first perform the computation $1.00000 \times 2^8 - 1.00000 \times 2^0$. The result of this expression is 1.1111111×2^7 which is unrepresentable in the number system, so a round-to-nearest operation is typically performed, returning 1.00000×2^8. This means that after the first iteration of the loop, the value of i would be the same as on entering the loop and as a result, the loop would never terminate.

```
float(4,5) i; // This notation implies i is a custom floating
point number with a 4 bit exponent and 5 bit mantissa.
while i > 0 do
    i = i − 1.0;
end while
```

Figure 8: Potentially non-terminating loop.

3.3 Termination provers

The importance of termination provers within the software verification community has led to many tools that can be used to prove termination. Amongst the techniques using ranking functions, a seminal contribution was the work of Podelski et al. [26] which is guaranteed to compute linear ranking functions for a loop body that

consists of linear arithmetic operations. This contribution formed the basis of more powerful tools, such as the TERMINATOR project, that perform more widespread program analysis to detect bugs in systems code [12, 27]. These tools automatically breaks down loops into the simplest possible structure, before using the method by Podelski to prove program termination of these individual loops.

However, from the perspective of numerical algorithms, there are two major limitations of reliance on the method by Podelski: a focus on linear arithmetic, and the absence of support for finite precision arithmetics such as floating point. Being able to model finite precision errors is imperative if we are to apply these tools to optimize the word-length of custom hardware designs, while nonlinear arithmetic operations, such as general multiplication, are common in most software programs.

There already exist several techniques that search for non-linear ranking functions for non-nested loops with non-linear loop transition statements [7, 14, 23]. However, these methods have exponential computational complexity in the number of program variables. To tackle this problem, we make use of scalable precision analysis techniques [1, 2] to check if a ranking function proves termination. Furthermore, we only apply this check for ranking functions that are either specified by a user or of a specific form. While this restriction limits the form of ranking functions that can be discovered, we believe this sacrifice is important for these techniques to be of use on real numerical algorithms. This is especially important since software is generally written in some form of finite precision arithmetic, and modelling finite precision arithmetic errors introduces many extra program variables.

Kittel [16] and Seneschal [11] are two tools that have attempted to incorporate fixed precision arithmetic using term re-write systems and SAT respectively. The analysis is performed at a bit-vector level, and this has the advantage that both techniques are able to consider bit-wise operations such as logical AND, OR and NOT. However, modelling variables at a bit-vector level is not scalable; for example, the number of constraints to represent multiplication using SAT grow quadratically with the word-length [28]. Furthermore, floating point operators are far more complicated than fixed point operators and the majority of algorithms in the field of scientific computing are written in floating point. Instead our approach uses standard safe over approximations from the numerical analysis community to model finite precision arithmetic at the word-level.

The only existing technique that would be applicable to floating point algorithms is bounded model checking. This unrolls the loop for a number of iterations and attempts to find an example of a recurring path within a while loop which demonstrates non-termination. For example, in Table 1, bounded model checking could detect the recurring value for $currAmount$, if it unrolled 21 iterations. However, while there have been substantial developments in speeding up bounded model checking using tools such as SAT, because this technique only explores a limited number unrolled iterations, it cannot prove termination in the general case.

In this paper, we seek to make these tools become more widely applicable in both software and hardware accelerator verification by introducing scalable techniques to deal with finite precision number systems numbers and improve the speed to which these tools handle all the basic arithmetic operations. In addition, we introduce how these techniques can be used to improve custom hardware designs.

4. BEYOND SSA

An overview of our approach is given in Figure 9. Firstly, our tool parses a program to extract the relevant information: the input

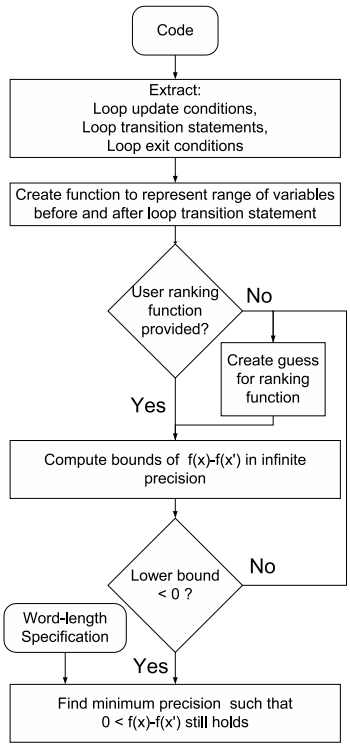


Figure 9: Overview of our approach.

variable ranges, loop exit conditions and loop transition statements. Using this information, we construct a representation of the ranges of the loop variables before and after an iteration of the loop body, taking into account finite precision errors. We then apply a chosen ranking function, this is either specified through a user hint, or our tool searches for a suitable ranking function. Provided the ranking function is valid, we proceed to find the minimum precision such that even in the worst-case, the chosen ranking functions will guarantee that the loop still terminates.

In this section, we outline first how we construct the model for the variables within the loop body. We then briefly discuss our technique to search for ranking functions using this model.

4.1 Representing the range of a variable

In this work, for floating point error, we have elected to use the multiplicative model of error used throughout numerical analysis literature. This represents the closest radix-2 floating-point approximation $\hat{x}$ to any real value x as in equation (1) [18], where η represents the number of mantissa bits used and δ represents the small unknown roundoff error. It is similarly possible to show that the radix-2 floating-point result of any scalar operation $(\odot)$ is bounded as in (2), provided the exponent is sufficiently large to span the range of the result, allowing us to create a polynomial to represent the potential range of a variable. In the example of Figure 8, the range of i' (after the loop update) could be represented as (3). For fixed point, we use a different model of error (4).

$$\hat{x} = x(1+\delta) \quad (|\delta| \leq 2^{-\eta}). \tag{1}$$

$$\widehat{x \odot y} = (x \odot y)(1+\delta). \tag{2}$$

$$i' = (i - 1.0)(1+\delta) \quad (|\delta| \leq 2^{-5}). \tag{3}$$

$$\hat{x} = x + \delta \quad (|\delta| \leq 2^{-\eta}). \tag{4}$$

4.2 Generating ranking functions

To prove termination, we must find a ranking function $f(x_1, ..., x_n)$ such that when it is applied to the loop variables before and after a loop transition, the ranking function will always decrease by more than some fixed amount (5). To represent the range of every loop variable before a loop update, we construct polynomials based upon information prior to entering the loop and on the loop conditions themselves. For the example in Figure 8, if the maximum value is specified to be 256, since the loop exit condition specifies $i > 0$, we could represent the range using equation (6). Similarly, we construct polynomials representing the ranges after the loop transition using the method described in the previous subsection. For the example in Figure 8, this would result in equation (7).

$$f(x_1', ..., x_n') < f(x_1, ..., x_n) - \epsilon \quad (5)$$

$$i \in 128 + 128i_1, \text{where } |i_1| \leq 1 \quad (6)$$

$$i' \in (127 + 128i_1)(1 + \delta), \text{where } |i_1| \leq 1, |\delta| \leq 2^{-5} \quad (7)$$

The next stage involves applying a ranking function to these polynomials to obtain $f(x')$ and $f(x)$. As we have described in Section 3.3, for algorithms that contain non-linear arithmetic operations, currently there are no scalable techniques that can automatically choose the ranking functions that can prove program termination. Indeed, our approach is only capable of checking if a ranking proves termination. Consequently, if no user-defined ranking function is supplied, our approach attempts to find a ranking function of the form described by (8) by exhaustively testing all the combinations of c_i until it finds a ranking function to prove termination in infinite precision.

$$f(x_1, ...,x_n) = c_0 + \sum_{i=1}^{n} c_i x_i \quad (8)$$

$$\text{where } c_0 \in \{-maxVal, 0, maxVal\},\ c_1, ...c_n \in \{-1, 0, 1\}$$

To prove termination with these functions, we perform a basic algebraic manipulation to equation (5) to create equation (9). This allows us to construct a single polynomial to which we can apply algebraic techniques to prove polynomial positivity [25] to determine whether the ranking function argument can prove termination. We apply the procedure described in [1, 2] because it satisfies these criteria whilst computing tight bounds in a scalable manner. These papers also highlighted that their procedure can make use of an iterative refinement to accommodate divisions in the input code and apply polynomial approximations to extend this technique for any elementary functions in the input code. This means we could apply our termination analysis to loops where transition statements are any of the basic algebraic operation or smooth elementary functions. Furthermore, this technique can check a termination proof is valid for any user specified ranking function provided that it can be expressed using a polynomial.

$$0 < f(x_1, ..., x_n) - f(x_1', ..., x_n') - \epsilon \quad (9)$$

If our approach is able to automatically find a ranking function and prove that it terminates using the above search technique, we can repeat this test with different choices of precision to find the minimum precision required to guarantee termination using this ranking function. If the search technique fails to find a ranking function that proves termination, it will require the user to provide a ranking function. Our program can then test if the provided ranking function will lead to termination and if successful, proceed to search for the minimum precision required to guarantee termination using this ranking function. In the example of Figure 8 where i lies in the range $[0;\ 256]$, if we use the ranking function $f(i) = i$, we would have to satisfy (10) to prove termination. As a result, it is trivial to see that the termination proof will depend on the input range and the chosen precision δ, as one would expect. We can compute that $|\delta| \leq 2^{-9}$ for this function to be positive.

$$0 < 128 + 128i_1 - (127 + 128i_1)(1 + \delta) - \epsilon \quad (10)$$

5. RESULTS

We demonstrate how our termination procedure can be used within a word-length optimization framework to improve a custom FPGA design using four case studies. We first discuss how to choose the fixed-point precision to guarantee termination for our half-life problem in Figure 4, and for a method to compute the greatest common divisor of two numbers. We then discuss two more complex numerical algorithms that require floating-point precision due to the high dynamic range of the variables. All these case studies contain loops with data dependent exit conditions where the loop transition statements contain multiplications and divisions. We also consider finite precision effects. Altogether, this takes it far beyond the scope of the tools described in Sections 2 and 3.3.

5.1 Case study: Half-life

The first stage of the process involves extracting the relevant loop information. Taking into account the user-specified input ranges, this is shown in Figure 10.

1: $decayRate \in [\frac{1}{16};\ \frac{15}{16}]$;
2: $limit \in [\frac{1}{16};\ \frac{15}{16}]$;
3: $years \in [1;\ 64]$;
4: $currAmount \in [\frac{1}{16};\ 1]$;
5: // Transition Statements
6: $years' = years + 1$
7: $currAmount' = currAmount \times decayRate$
8: Terminates if:
$0 < f(years, currAmount) – f(years', currAmount') – \epsilon$

Figure 10: Halflife algorithm broken down into transition test.

The next stage of the process involves choosing the ranking function; the search method described in Section 4.2 will find the ranking function $f(years, currAmount) = currAmount$. First, we test that this is a valid ranking function. For this to be the case, $f(years, currAmount) > 0$. This is trivially the case since we have specified $currAmount$ to lie in the range $[1/16; 1]$. As a result, after applying the model of fixed point error described by equation (4), our procedure will then attempt to bound the function (11), for different values of precision δ. Figure 11 shows the computed lower bound for this function.

$$currAmount - (currAmount \times decayRate + \delta) \quad (11)$$

This lower bound crosses 0 when 8 bits are used. This means that when 8 bits are used for the variable $currAmount$, it is guaranteed to terminate. For this simple example, we can test that this result is correct by hand. The worst case is when $limit$ is $1/16$ and $decayRate$ is $15/16$. Suppose in this case, $currAmount$ could eventually decrease to the value $1/16$. The result for $1/16 \times 15/16 =$

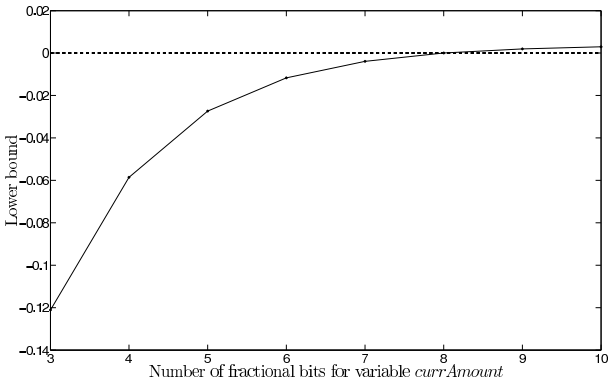


Figure 11: Computed lower bound of $f(currAmount) - f(currAmount')$ using the approach described in [1, 2]. For a given precision, if the lower bound is above zero it is guaranteed to terminate.

$^{15}/_{256}$ in binary is 0.00001111. With 7 bits, this would round up to 0.000100, which is $^{1}/_{16}$, so this would never terminate. However, with 8 bits, it could exactly represent 0.00001111, so it would terminate.

Impact on FPGA Implementation: In order to quantify how much hardware could be saved by making use of this analysis, we have hand-coded a custom FPGA-based accelerator for this algorithm using VHDL according to the architecture of Figure 5. Table 2 shows the post-place and route resource use for different choices of precision for this hardware implementation, targeted to a Virtex 7. We can see that even on this trivial example, it is desirable to choose the minimum precision required to guarantee termination because this design achieves a higher clock frequency and uses fewer resources in terms of slices and DSPs. Furthermore, the reduced silicon area use will result in lower power consumption.

Table 2: Resource use and max frequency of custom hardware implementations of halflife example.

# bits for $currAmount$	Slices	DSPs	Frequency (MHz)
8	14	1	275
12	15	1	275
16	16	1	275
20	17	2	210

We note that with the exception of exhaustive simulation, none of the existing word-length optimization tools described in Section 2 could be used to determine the precision for $currAmount$. Similarly, none of the termination provers described in Section 3.3 could prove that the smallest design that is guaranteed to terminate. For example, due to the multiplication, Kittel [16] and Seneschal [11] both reach time-outs when attempting to compute a result. Furthermore, exhaustive simulation is only applicable because the inputs to this example only consist of four fractional bits. If the inputs required were described by many more bits, exhaustive simulation would quickly become infeasible. In contrast, because our analysis involves bounding a polynomial (11) and the input range would not affect the size of this polynomial, our approach would compute the minimum precision required to guarantee termination for an example with a wider input range within the same execution time.

In the following case studies, we will study increasingly complex examples where only our approach could compute the minimum precision required to guarantee termination within a tractable amount of time.

5.2 Case study: Euclidean Division

Our second case study is Euclid's method to compute the greatest common divisor of two numbers. This algorithm is given in Figure 12. Once again, we shall step through the various steps of our framework. After extracting the relevant loop information, we obtain the information in Figure 13.

```
1: // Algorithm to compute the GCD of two numbers a and b,
2: // where a lies in the interval [0.1; 1000], b lies in the interval [0.1; 100] and a > b.
3: Function GCD(a, b)
4: while b > 0.1 do
5:     c = b;
6:     b = a − ⌊a/b⌋b
7:     a = c;
8: end while
9: return(a)
```

Figure 12: Euclid's method to compute the greatest common divisor of two variables.

```
1: a ∈ [0.1; 1000];
2: b ∈ [0.1; 100];
3: c ∈ [0.1; 1000];
4: // Transition Statements
5: c' = b
6: b' = a − ⌊a/b⌋ × b
7: a' = b
8: Terminates if:
   0 < f(b) − f(b') − ϵ
```

Figure 13: Euclid's method broken down into transition test.

Once again, the search can be used to select the ranking function, in this case it will find the function $f(a, b, c) = b$. This can trivially be shown to be a valid ranking function because b lies in the range $[0.1;\ 100]$. In order to apply the polynomial techniques to bound the function, we first must approximate the floor function. To do this, we note that the floor function has a limited range as shown in (12), so we bound this worst case range using (13). We can then once again apply the model of fixed point error described in equation (4), and attempt to compute the lower bound of the function (14), for different values of precision δ_i. This is shown in Figure 14.

$$^{a}/_{b} - 1 \leq \lfloor {}^{a}/_{b} \rfloor \leq {}^{a}/_{b} \tag{12}$$

$$^{a}/_{b} - 1 \leq {}^{a}/_{b} - 0.5 + y \leq {}^{a}/_{b}, \text{where } |y| \leq 0.5 \tag{13}$$

$$0 < b - (a - (({}^{a}/_{b} + \delta_1) - 0.5 + y) \times b + \delta_2) \tag{14}$$

$$\text{where } |y| \leq 0.5 \text{ and } |\delta_i| \leq 2^{-\eta}$$

Impact on FPGA Implementation: Figure 15 illustrates a custom FPGA-based accelerator for this algorithm, which we have hand-coded VHDL. For this implementation, using our approach we can see that the lower bound crosses 0 in Figure 14 when 5 fractional bits are used. This implies that to guarantee termination a total of 15 bits (10 integer bits are needed represent the range between 0 and 1000) are required for the variable b as well as all the intermediate variables that are used to compute b. The post-place

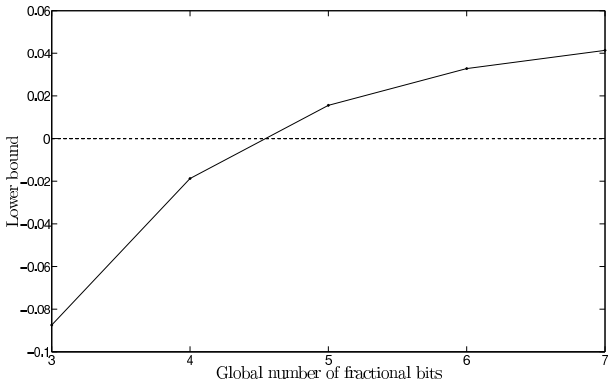


Figure 14: Computed lower bound of $f(b) - f(b')$.

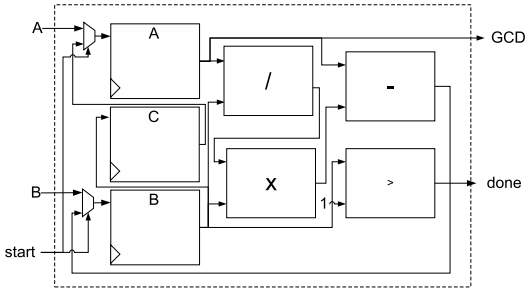


Figure 15: Hardware accelerator for Euclid's method to find the greatest common divisor.

and route resource use for different choices of precision for this example are shown in Table 3. Whereas the last example consisted of only a multiplier, counter, comparator and some registers, this example also contains a subtracter and a divider. As a result, it becomes of even greater importance to choose the minimum precision. Using a single additional fractional bit requires an increase in slices of 30%; using 5 additional fractional bits requires 90% more slices and 2 times the number of DSPs. This is mainly because the divider, created using Xilinx Coregen, consumes substantially more resources with increasing number of bits. This highlights the importance of ensuring the analytical techniques described in this work are available when performing word-length optimization.

Table 3: Resource use and max frequency of custom hardware implementations of Euclid's method to compute the greatest common divisor of two variables.

# fractional bits	Slices	DSPs	Frequency (MHz)
5	141	1	230
6	183	1	225
7	203	1	220
8	262	1	190
9	289	1	195
10	274	2	180

5.3 Case study: Newton's Method

Our third case study analyzes how our approach could used when trying to create a dedicated hardware accelerator to compute the square root of any number lying in the interval $[0;\ 100]$ using Newton's Method, as described in Figure 16.

For this case study, after extracting the loop information, we add a hint that incorporates the fact that the loop will never be entered when $0.9999999 < i < 1.0000001$. This hint tells the tool to explore regions either side of this range separately. This results in the two tests given in Figure 17, where in the second test we also take into account the condition $|i_{k+1} - i_k| > \eta$ to limit the lower bound. Once it is in this form, our tool applies the floating point model of error described in Section 4.1, to represent the ranges of all the intermediate variables (i_sq, num, den and i'). Finally, our search procedure finds the ranking functions $f(i) = i$ for the first test, and $f(i) = maxFloat - i$ for the second test. We use this information to make a custom FPGA accelerator.

1: // Algorithm to find square root of any input number i to a tolerance η,
2: // where i lies in the interval $[0;\ 100]$ and $\eta = 1 \times 10^{-7}$
3: Function Newton_SQRT(i)
4: $i_0 = 1, i_1 = i; k = 0$
5: **while** $|i_{k+1} - i_k| > \eta$ **do**
6: $\quad i_{k+1} = \frac{1}{2}\left(i_k + \frac{i}{i_k}\right)$
7: $\quad k = k + 1;$
8: **end while**
9: return(i)

Figure 16: Newton method to compute the square root of a number.

1: // First test	1: // Second test
2: $i \in [1.0000001;\ 100]$;	2: $i \in [0.0000001; 0.9999999]$;
3: $\eta = 1 \times 10^{-7}$;	3: $\eta = 1 \times 10^{-7}$;
4: // Transition Statements	4: // Transition Statements
5: $i\_sq = i \times i$	5: $i\_sq = i \times i$
6: $num = i\_sq + i$	6: $num = i\_sq + i$
7: $den = 2 \times i$	7: $den = 2 \times i$
8: $i' = \frac{num}{den}$	8: $i' = \frac{num}{den}$
9: Terminates if:	9: Terminates if:
$f(i') < f(i) - \epsilon$	$f(i') < f(i) - \epsilon$

Figure 17: Newton method broken down into transition tests.

Impact on FPGA Implementation: A parallel FPGA implementation of this square root circuit is shown in Figure 18. In this example, using the same process, we can calculate that at least 27 bits (or a precision of 7.5×10^{-9}) are required to guarantee the termination. Table 4 demonstrates the resource use of this circuit using the minimum precision necessary as found by our technique, as well as using IEEE-754 standard single or double precision. We see that our technique can create a circuit that uses over 50% less silicon area than IEEE-754 standard double precision. This could in turn be used to increase the level of parallelism to create a faster square root circuit; a simple technique to achieve this would be to unroll the loop until all the area of the FPGA is consumed.

Table 4: Resource use and max frequency of custom hardware implementations to find square roots using Newton's Method.

Method	Precision (# bits)	Slices	Frequency (MHz)
Our Approach	27	1,236	490
IEEE Single Precision	24	999	500
IEEE Double Precision	53	2,667	330

While one may be tempted to use IEEE-754 single precision throughout a custom hardware implementation as this requires even less silicon area, because we are unable to prove termination in the

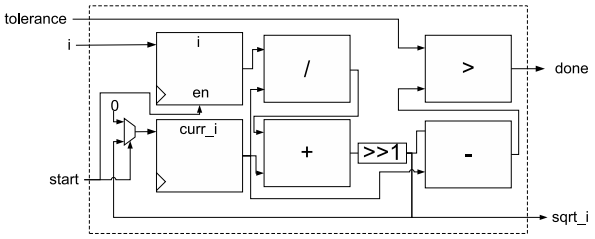


Figure 18: Hardware accelerator for Newton's method to compute the square root.

case of round-off errors, such a design may not be safe to use. Furthermore, in a software setting, this highlights how our technique could also be used to locate potential termination bugs; indeed, when running the code of Figure 16 in single precision in Matlab with an input of $i = 1.01$, the loop fails to terminate after 1 minute of execution time. In contrast, the execution time to check if termination is guaranteed for a given precision, was less than 20 seconds using our approach, and it generates a safe implementation that is guaranteed to work.

5.4 Case study: Adaptive Euler's Method

The final case study is Euler's Method, which is a numerical method to approximate the solution to a differential equation, described in Figure 19. This example differs from the previous examples in that it has two loops and the latter is a repeat loop. However, the repeat loop can be re-written into a while loop, and provided we treat each loop separately, the same process can be applied to prove termination of the algorithm.

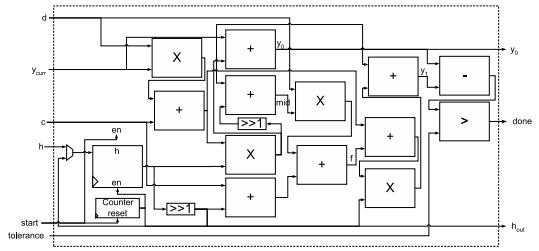
/* Algorithm to solve differential equation of the form $y = c + dy$, where the initial value y_{in} lies in the interval $[1000;\ 1100]$. The remaining input variables are bounded as follows:
$a \in [0;\ 1]$ and $b \in [0;\ 1]$, $c \in [90;\ 100]$, $d \in [0.4;\ 0.5]$, $\eta_1 \in [0.1;\ 1]$ and $\eta_2 \in [10^{-6};\ 1]$*/

1: Function Euler($a, b, c, d, y_{in}, \eta_1, \eta_2$)
2: $y_{curr} = y_{in}$
3: **while** $a < b$ **do**
4: $h = 0.5$
5: **do**
6: $y_0 = y_{curr} + h \times (c + d \times y_{curr})$
7: $mid = y_{curr} + \frac{1}{2}h \times (c + d \times y_{curr})$
8: $f = c + \frac{1}{2}h + d \times mid$
9: $y_1 = y_{curr} + \frac{1}{2}h \times ((c + d \times y_{curr}) + f)$
10: $\tau = y_1 - y_0$
11: $h = \frac{1}{2}h$
12: **while** $(\tau > \eta_1)\&\&(h > \eta_2)$
13: $y_{curr} = y_0$;
14: $a = a + 2 \times h$;
15: **end while**
16: return(a)

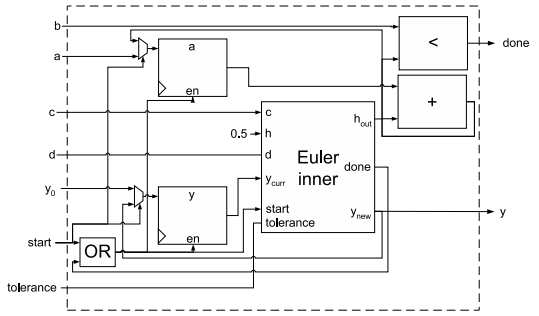
Figure 19: Euler's Method for Solving Differential Equations.

Once again, the search procedure will select the ranking functions, for the inner loop this will be $f(\tau) = \tau$ and for the outer loop $f(a) = 1000 - a$.

Impact on FPGA Implementation: After applying our bounding procedure, we can compute that a 9 bit precision is required to guarantee termination of the inner loop and 17 bits for the outer loop. We note that we could use these different precisions within the inner and outer loops in a custom hardware design. If we were to implement the hardware using 9 bits for every operator in the inner loop and 17 bits for the outer loop, we would create the hardware described in Table 5. In this case, it would save 50% or 80% of the silicon area of an IEEE-754 standard single or double precision arithmetic implementation respectively.



(a) Inner Loop.



(b) Outer loop.

Figure 20: Hardware accelerator for Euler's method for Solving Differential Equations.

Table 5: Resource use and max frequency of custom hardware implementations of Euler's Method for Solving Differential Equations.

Method	Precision (# bits)	Slices	Frequency (MHz)
Our Approach	9,17	418	550
IEEE Single Precision	24	926	490
IEEE Double Precision	53	2230	480

6. CONCLUSION

This paper has discussed a novel technique that can be used to determine the minimum precision throughout a loop with data dependent exit conditions. We have shown that we can use it to design safe hardware that is guaranteed to terminate, and this can result in silicon area savings of up to 80% over a naïve approach of using IEEE-754 standard double precision arithmetic. No existing analytical technique could be used to provide such guarantees. Furthermore, the third case study also demonstrated the potential of our work to locate the presence of bugs in software programs. In the future, we wish to refine the techniques described in this paper to make it applicable to more sophisticated algorithms as well as to analyze the relationship between precision and convergence.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the EPSRC (Grant EP/I020357/1 and EP/I012036/1) and the EU FP7 project REFLECT.

8. REFERENCES

[1] D. Boland and G. Constantinides. Bounding variable values and round-off effects using Handelman representations. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1691 –1704, 2011.

[2] D. Boland and G. A. Constantinides. A scalable approach for automated precision analysis. In *Proc. Int. Symp. on Field Programmable Gate Arrays*, pages 185–194, 2012.

[3] G. Caffarena, C. Carreras, J. A. López, and A. Fernández. SQNR estimation of fixed-point DSP algorithms. *EURASIP J. Adv. Signal Process*, 2010:21:1–21:12, 2010.

[4] M.-A. Cantin, Y. Savaria, and P. Lavoie. A comparison of automatic word length optimization procedures. In *Proc. Int. Symp. on Circuits and Systems*, volume 2, pages II–612 – II–615 vol.2, 2002.

[5] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie. An automatic word length determination method. In *Proc. Int. Symp. on Circuits and Systems*, pages 53 –56 vol. 5, 2001.

[6] M. L. Chang and S. Hauck. Automated least-significant bit datapath optimization for FPGAs. *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 59–67, 2004.

[7] Y. Chen, B. Xia, L. Yang, N. Zhan, and C. Zhou. Discovering non-linear ranking functions by solving semi-algebraic systems. In *Proc. Int. Conf. on Theoretical Aspects of Computing*, pages 34–49, 2007.

[8] G. Constantinides, P. Cheung, and W. Luk. The multiple wordlength paradigm. In *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, pages 51 –60, 2001.

[9] G. Constantinides, P. Cheung, and W. Luk. Optimum wordlength allocation. *Proc. Int. Symp. Field-Programmable Custom Computing Machines*, pages 219–228, 2002.

[10] G. Constantinides, A. Kinsman, and N. Nicolici. Numerical data representations for FPGA-based scientific computing. *IEEE Design Test of Computers*, 28(4):8–17, 2011.

[11] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250, 2010.

[12] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. Conf. on Programming language design and implementation*, pages 415–426, 2006.

[13] B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, May 2011.

[14] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *Proc. Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 1–24, 2005.

[15] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Computers.*, 60(2):242–253, 2011.

[16] S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *Proc. Int. Conf. on Verified Software: Theories, Tools, Experiments*, pages 261–277, 2012.

[17] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In *Proc. Int. Conf. on Field-Programmable Logic and Applications*, pages 523–533, London, UK, 2002.

[18] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Soc for Industrial & Applied Math, Philadelphia, PA, USA, second edition, 2002.

[19] A. B. Kinsman and N. Nicolici. Robust design methods for hardware accelerators for iterative algorithms in scientific computing. In *Proc. Design Automation Conference*, pages 254–257, 2010.

[20] K.-I. Kum and W. Sung. Combined word-length optimization and high-level synthesis of digital signal processing systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):921–930, Aug 2001.

[21] D.-U. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1990–2000, Oct. 2006.

[22] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk. Minibit: bit-width optimization via affine arithmetic. In *Proc. Design Automation Conference*, pages 837–840, New York, NY, USA, 2005. ACM.

[23] Y. Li. Automatic discovery of non-linear ranking functions of loop programs. In *Proc. Int. Conf. on Computer Science and Information Technology*, pages 402 –406, 2009.

[24] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Int. Conf. on Field Programmable Logic and Applications*, pages 617–620, Aug. 2007.

[25] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming Ser. B*, 96(2-3):293–320, 2003.

[26] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. int. conf. on Verification, Model Checking, and Abstract Interpretation*, pages 239–251, 2004.

[27] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *Proc. Int. Conf. on Practical Aspects of Declarative Languages*, pages 245–259, 2007.

[28] A. Sülfow, U. Kühne, R. Wille, D. Große, and R. Drechsler. Evaluation of sat like proof techniques for formal verification of word level circuits. In *Proc. Workshop on RTL and High Level Testing.*, pages 31Ű–36, 2007.

[29] W. Sung and K.-I. Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Trans. on Signal Processing*, 43(12):3087–3090, 1995.